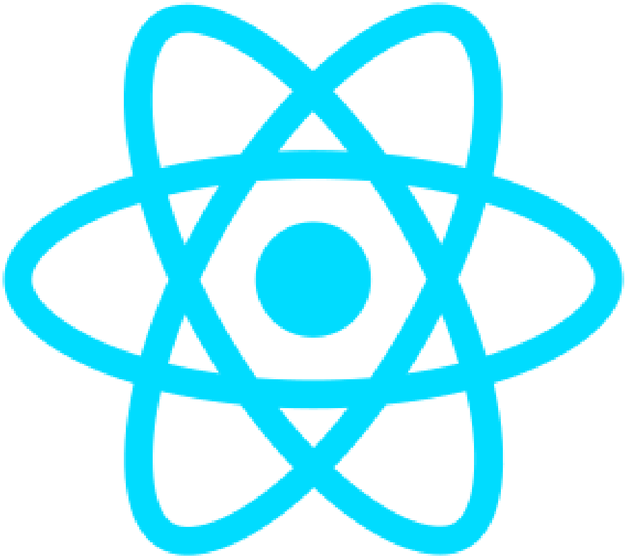
React JS

Partie 1 : Concepts de base

JSX : Syntaxe et fonctionnement

Qu’est-ce que JSX ?

- JSX (**JavaScript XML**) est une **extension de syntaxe de JavaScript**, utilisée dans React pour décrire l'interface utilisateur.
- Il permet de mélanger du code **HTML-like** avec du **JavaScript**, facilitant la construction d’interfaces complexes en React.
- Par exemple, un élément JSX ressemble à du HTML mais est converti en appels de fonctions JavaScript lors de la compilation.

```
const element = <h1>Hello, world!</h1>;
```

Partie 1 : Concepts de base

Différence avec le HTML

JSX n’est pas du HTML pur. Bien que la syntaxe soit similaire, il y a quelques différences clés :

Balises fermées: En JSX, toutes les balises **doivent être fermées**, même celles qui sont **auto-fermantes**.

```html
<img src="image.png">
```

HTML

```jsx
<img src="image.png" />
```

JSX

Attributs en camelCase: En HTML, on utilise des attributs en minuscule, alors qu'en JSX, on les écrit en **camelCase**.

```html
<button onclick="handleClick()">Click me</button>
<div class="container"></div>
```

HTML

```jsx
<button onClick={handleClick}>Click me</button>
<div className="container"></div>
```

JSX

Expressions JavaScript intégrées: Dans JSX, on peut intégrer des expressions JavaScript en utilisant des accolades **{}**.

```jsx
const name = "John";
const element = <h1>Hello, {name}!</h1>;
```

JSX

Dans cet exemple, la variable name est intégrée directement dans le JSX, et l’élément rendu affichera "Hello, John!".

Partie 1 : Concepts de base

Composants

Création d’un composant fonctionnel

Composants fonctionnels : Ce sont des fonctions JavaScript qui retournent du JSX. Ils permettent de structurer l'application en petites unités réutilisables. Voici un exemple simple :

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Ici, Welcome est un composant fonctionnel qui prend des props en paramètre et retourne du JSX. Chaque composant doit commencer par une majuscule.

Composition et réutilisation des composants

```
function App() {
  return (
    <div>
      <Welcome name="Alice" />
      <Welcome name="Bob" />
      <Welcome name="Charlie" />
    </div>
  );
}
```

Une des forces de React est la capacité de composer des composants entre eux. Un composant peut en contenir d’autres, ce qui permet de structurer l’interface utilisateur de manière modulaire :

Ici, le composant App réutilise trois fois le composant Welcome avec des props différentes.

Partie 1 : Concepts de base

Composants

Props : Passage de données aux composants

Props (propriétés) sont un moyen de passer des données d’un composant parent à un composant enfant. Les props sont immuables, ce qui signifie qu'un composant ne peut pas les modifier :

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Ici, le prop name est utilisé pour personnaliser le contenu du composant.

```
<Welcome name="Alice" />
```

Partie 1 : Concepts de base

Composants

Exemple complet : Composant avec JSX et props

Voici un exemple complet d'une petite application React qui utilise JSX, des composants fonctionnels, et des props :

```
import React from 'react';
import ReactDOM from 'react-dom';

function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Alice" />
      <Welcome name="Bob" />
      <Welcome name="Charlie" />
    </div>
  );
}

ReactDOM.render(<App />, document.getElementById('root'));
```

Dans cet exemple :
Welcome est un composant qui reçoit une prop et affiche un message personnalisé.
App est le composant principal qui affiche trois composants Welcome avec des noms différents.

Partie 2 : Gestion d'état

Les hooks

Un hook dans React est une fonction spéciale qui permet d'**utiliser des fonctionnalités** propres à React, comme la gestion de l'état ou les effets secondaires, à l'intérieur des composants fonctionnels. Avant l'introduction des hooks dans la version 16.8 de React, certaines fonctionnalités comme le state ou le cycle de vie d'un composant n'étaient accessibles qu'à travers des composants de classe. Les hooks ont simplifié cette structure en rendant ces fonctionnalités disponibles dans des composants fonctionnels.

Principales caractéristiques des hooks :

- Sans classe : Les hooks permettent de gérer des fonctionnalités complexes, comme l'état ou le cycle de vie, dans des composants fonctionnels, évitant ainsi l'utilisation des classes.
- Réutilisabilité : Grâce aux hooks, vous pouvez réutiliser de la logique entre différents composants sans avoir à manipuler des classes ou des méthodes complexes.
- Encapsulation de la logique : Les hooks peuvent être utilisés pour encapsuler une logique (comme la gestion de l'état ou un effet) que vous pouvez réutiliser dans plusieurs composants en créant des hooks personnalisés.

Les hooks les plus utilisés : useState, useEffect, useContext

En plus des hooks fournis par React, il est possible de créer ses propres hooks personnalisés pour encapsuler des morceaux de logique réutilisables. Ces hooks personnalisés permettent de partager facilement des fonctionnalités entre différents composants sans avoir à dupliquer le code.

Partie 2 : Gestion d'état

Les hooks

En plus des hooks fournis par React, il est possible de créer ses propres hooks personnalisés pour encapsuler des morceaux de logique réutilisables. Ces hooks personnalisés permettent de partager facilement des fonctionnalités entre différents composants sans avoir à dupliquer le code.

Comment créer un hook personnalisé ?
Un hook personnalisé n'est qu'une fonction JavaScript normale dont le nom commence par "use" et qui peut utiliser d'autres hooks de React à l'intérieur. Il suit les mêmes règles que les hooks standard.
Exemple de hook personnalisé :
Créons un hook personnalisé qui gère une minuterie.

```jsx
import { useState, useEffect } from 'react';

// Hook personnalisé pour une minuterie
function useTimer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds((prevSeconds) => prevSeconds + 1);
    }, 1000);

    // Nettoyage de l'intervalle lorsque le composant est démonté
    return () => clearInterval(interval);
  }, []);

  return seconds;
}

export default useTimer;
```

```jsx
import React from 'react';
import useTimer from './useTimer';

const TimerComponent: React.FC = () => {
  const seconds = useTimer();  // Utilise le hook personnalisé

  return <div>Compteur : {seconds} secondes</div>;
};

export default TimerComponent;
```

Points importants sur les hooks personnalisés :

- Un hook personnalisé doit toujours commencer par "use" (comme useTimer, useAuth, etc.) pour que React puisse comprendre qu'il s'agit d'un hook.
- Vous pouvez utiliser des hooks standards (comme useState et useEffect) dans vos hooks personnalisés.
- Les hooks personnalisés ne sont qu'un moyen de réutiliser la logique entre les composants. Ils n'ajoutent pas de nouvelles fonctionnalités à React, mais simplifient le code.

Partie 2 : Gestion d'état

Le Hook useState

Comprendre le state dans React

- Le state (état) dans React est un objet qui représente les données dynamiques d'un composant. Contrairement aux props, qui sont passées par les composants parents, le state est **local** au composant et peut être modifié directement par ce dernier. Il permet d'ajuster le rendu du composant en fonction des interactions de l'utilisateur ou d'autres événements.
- Pour gérer cet état de manière déclarative, React utilise le Hook useState. Ce Hook permet de définir et de modifier l'état interne d'un composant fonctionnel.

Utilisation de useState en TypeScript

- En TypeScript, l'utilisation de useState nécessite souvent de définir le type de l'état, car TypeScript impose une typage statique. Vous pouvez laisser TypeScript inférer le type, ou le déclarer explicitement.
- Voici comment utiliser useState avec du typage en TypeScript :

Exemple 1 : Utilisation basique de useState avec typage

```tsx
import React, { useState } from 'react';

const Counter: React.FC = () => {
  // Le type est inféré ici, car la valeur initiale est un nombre.
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Vous avez cliqué {count} fois</p>
      <button onClick={() => setCount(count + 1)}>Cliquez ici</button>
    </div>
  );
};
```

Partie 2 : Gestion d'état

Le Hook useState

La syntaxe utilisée pour déclarer l'état avec le hook useState dans React, comme dans [count, setCount], peut sembler un peu déroutante au début. Cependant, elle s'appuie sur une technique native de JavaScript appelée "**destructuration** de tableau".

- Le premier élément est **la valeur actuelle de l'état.**
- Le deuxième élément est une **fonction pour mettre à jour cet état**.

```
const [count, setCount] = useState(0);
```

La fonction **setCount** vous permet de **mettre à jour la valeur de l'état** count. Lorsqu'elle est appelée, **React re-rend (redraw)** le composant avec la nouvelle valeur d'état. Ce mécanisme permet de créer des interfaces dynamiques et interactives.
Sans cette fonction, il serait impossible de modifier directement la valeur de l'état.
C'est important, car l'état est **immuable** : vous ne pouvez pas le modifier directement, vous devez utiliser la fonction fournie pour ce faire.

Partie 2 : Gestion d'état

Le Hook useState

Exemple 2 : Spécification explicite du type

```tsx
import React, { useState } from 'react';

const MyComponent: React.FC = () => {
  // Ici, l'état peut être une chaîne ou null.
  const [name, setName] = useState<string | null>(null);

  return (
    <div>
      <input
        type="text"
        value={name || ''}
        onChange={(e) => setName(e.target.value)}
        placeholder="Entrez votre nom"
      />
      {name && <p>Bonjour, {name}!</p>}
    </div>
  );
};
```

Parfois, il peut être nécessaire de déclarer explicitement le type de l'état, surtout lorsque la valeur initiale est null ou peut changer de type.

Dans cet exemple, useState<string | null> signifie que l'état name peut être soit une chaîne de caractères (string), soit null.

Partie 2 : Gestion d'état

Le Hook useEffect

Gestion des effets secondaires

useEffect est un hook qui permet d'exécuter du code à des moments spécifiques du cycle de vie d'un composant. C'est un excellent moyen de gérer les effets secondaires, comme la récupération de données à partir d'une API, la modification du DOM, ou la gestion des abonnements.
Il fonctionne de manière similaire à componentDidMount, componentDidUpdate, et componentWillUnmount des composants de classe. Le hook useEffect prend deux arguments : une fonction à exécuter et un tableau de dépendances.

Exemple 1 : useEffect simple
Cet exemple montre un composant qui change le titre de la page en fonction de l'état local count.

Dans cet exemple :

- Le hook useEffect est déclenché chaque fois que la variable count est mise à jour.
- Le tableau des dépendances [count] indique que cet effet ne sera exécuté que lorsque count change. Si le tableau est vide [], l'effet ne s'exécute qu'une seule fois, lors du montage du composant (équivalent à componentDidMount).

```tsx
import React, { useState, useEffect } from 'react';

const CounterWithEffect: React.FC = () => {
  const [count, setCount] = useState<number>(0);

  // Exécution de cet effet à chaque fois que 'count' change
  useEffect(() => {
    document.title = `Vous avez cliqué ${count} fois`;
  }, [count]); // Le tableau des dépendances inclut 'count'

  return (
    <div>
      <p>Vous avez cliqué {count} fois</p>
      <button onClick={() => setCount(count + 1)}>Cliquez ici</button>
    </div>
  );
};
```

Partie 2 : Gestion d'état

Le Hook useEffect

Exemple 2 : useEffect avec nettoyage
Il est souvent nécessaire de nettoyer un effet pour éviter des fuites de mémoire, surtout lorsqu'on gère des abonnements, des timers ou des requêtes.

Dans cet exemple :

- Un intervalle est défini pour incrémenter l'état seconds toutes les secondes.
- Le retour de la fonction dans useEffect permet de nettoyer l'intervalle lorsque le composant est démonté, afin d'éviter des fuites de ressources.

```tsx
import React, { useState, useEffect } from 'react';

const TimerComponent: React.FC = () => {
  const [seconds, setSeconds] = useState<number>(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(prev => prev + 1);
    }, 1000);

    // Nettoyage de l'effet lorsque le composant est démonté
    return () => clearInterval(interval);
  }, []); // Effet exécuté uniquement une fois au montage

  return <div>Compteur : {seconds} secondes</div>;
};
```

Partie 3 : Interaction utilisateur

Gestion des événements avec TypeScript
Dans React, les événements fonctionnent de manière similaire aux événements JavaScript standards, mais ils utilisent une syntaxe spécifique et une gestion améliorée via JSX et les hooks. En TypeScript, il est important de typer les événements pour assurer la sécurité du code et bénéficier de l'auto-complétion.
Exemple de gestion d'un événement en TypeScript
Voici comment gérer un événement de clic sur un bouton en utilisant TypeScript :

Dans cet exemple :
L'événement de clic onClick est associé à la fonction handleClick.
Le type de l'événement est React.MouseEvent<HTMLButtonElement>, qui spécifie qu'il s'agit d'un événement de souris déclenché sur un bouton.

```tsx
import React from 'react';

const ButtonClickExample: React.FC = () => {
  const handleClick = (event: React.MouseEvent<HTMLButtonElement>) => {
    console.log("Button clicked", event);
  };

  return <button onClick={handleClick}>Click Me</button>;
};

export default ButtonClickExample;
```

Partie 3 : Interaction utilisateur

Événement de formulaire (submit) :

```
const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {
  event.preventDefault();
  console.log("Form submitted");
};
```

Événement d'entrée utilisateur (input change) :

```
const handleInputChange = (event: React.ChangeEvent<HTMLInputElement>) => {
  console.log("Input value: ", event.target.value);
};
```

Ateliers

1. Application de gestion des dépenses (picsouApp)

Description : Créez une application qui permet aux utilisateurs de suivre leurs dépenses et de voir un récapitulatif.

Fonctionnalités:
Ajout et suppression de dépenses
Calcul du total des dépenses
Catégorisation des dépenses (alimentaire, transport, etc.)